Realtime Ray Tracing on GPUs: NVIRT by NVIDIA and RayGLSL by LFK

Andreas Maier (Dipl.-Inf (FH))

Advanced Seminar
Seminar Thesis Summer 2009

Master of Science
Computer Graphics and Image Processing
Advisor: Prof. Dr. Alfred Nischwitz
# Contents

1 Introduction ................................. 2
   1.1 Overview .................................. 2
   1.2 Motivation .............................. 2

2 Ray Tracing ................................. 4
   2.1 The Algorithm .......................... 4
   2.2 Acceleration Structures ............... 5
      2.2.1 Uniform Grids .......................... 5
      2.2.2 BVHs ................................ 6
      2.2.3 KD-Trees ............................ 6
   2.3 Caustic Graphics ......................... 7
      2.3.1 CausticRT Overview .................. 7
      2.3.2 API, Design and Hardware .......... 8
      2.3.3 Possible Drawbacks .................. 9
      2.3.4 Prospect ............................ 10
   2.4 NVIDIA’s NVIRT ......................... 10
      2.4.1 About NVIRT ......................... 10
      2.4.2 NVIRT Overview ..................... 11
      2.4.3 The NVIRT API ...................... 12
         2.4.3.1 The Elements of NVIRT .......... 12
         2.4.3.2 Mode of Operation of an NVIRT Application ... 15
   2.5 RayGLSL ................................. 17
      2.5.1 RayGLSL and SLI ..................... 18
      2.5.2 Optimization Possibilities of RayGLSL ... 19
         2.5.2.1 Overview of the proposed Algorithms ... 19
         2.5.2.2 Stackless KD-Tree Traversal for High Performance GPU ... 20
      2.5.3 Implementing Billboards and Alpha Textures into RayGLSL ... 25
      2.5.4 Prospect and Discussion of the Proposed Optimization Techniques ... 26

3 Conclusions ................................. 27
   3.1 Summary .................................. 27
   3.2 Final thoughts ........................... 27
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The ray tracing algorithm</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>Acceleration structures for ray tracing l.t.r. Uniform Grid, Bounding Volume Hierarchy and KD-Tree</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>The structure of CausticRT</td>
<td>8</td>
</tr>
<tr>
<td>2.4</td>
<td>The CausticOne card</td>
<td>9</td>
</tr>
<tr>
<td>2.5</td>
<td>Images rendered with CausticRT</td>
<td>10</td>
</tr>
<tr>
<td>2.6</td>
<td>Scenes from NVIDIA’s GPU real time ray tracer presented at SIGGRAPH08</td>
<td>11</td>
</tr>
<tr>
<td>2.7</td>
<td>The elements of the NVIRT API</td>
<td>13</td>
</tr>
<tr>
<td>2.8</td>
<td>Der Ablauf in die parallelisierbaren Stufen eines GPU Ray Tracers</td>
<td>13</td>
</tr>
<tr>
<td>2.9</td>
<td>The nvirt ray tracing workflow</td>
<td>15</td>
</tr>
<tr>
<td>2.10</td>
<td>One of the first scenes generated using ray tracing. This one is created using nvirt</td>
<td>16</td>
</tr>
<tr>
<td>2.11</td>
<td>Simplified workflow of the whited stile ray tracing application in nvirt</td>
<td>16</td>
</tr>
<tr>
<td>2.12</td>
<td>Object hierarchies used in the whitted scene</td>
<td>17</td>
</tr>
<tr>
<td>2.13</td>
<td>The workload distribution for one frame on RayGLSL</td>
<td>19</td>
</tr>
<tr>
<td>2.14</td>
<td>KD-Tree with ropes</td>
<td>21</td>
</tr>
<tr>
<td>2.15</td>
<td>The entry point criterion</td>
<td>22</td>
</tr>
<tr>
<td>2.17</td>
<td>The if-if trace() method</td>
<td>24</td>
</tr>
<tr>
<td>2.16</td>
<td>The while-while trace() method</td>
<td>24</td>
</tr>
</tbody>
</table>
Abstract

Currently everyone is speaking about ray tracing. In 1968 first steps in the fields of Ray Tracing were made. Since then, this topic keeps appearing in the media continuously. However, until now there are only applications in the high quality offline area. In the real-time sector it has been kept in check consequently by the rasterization\footnote{http://en.wikipedia.org/wiki/Rasterization} algorithm, which current graphics cards are designed for. But with increasing content in such applications, there's a call for a more flexible illumination model showing up. But until now there have not been any adequate candidates in the hardware area, as well as on the software area. But since Intel has started its search for a new killer app for its upcoming graphics platform Larabee, the words real-time and ray tracing have been connected an increasing number of times. Within their search they came up with the idea to avoid current GPU limitations by a more general streaming pipeline, by using their x86 architecture for a stream processor. But it is not the intention of Intel\footnote{General Purpose Computation on GPUs}, NVIDIA\footnote{NVIRT API\cite{29} for NVIDIA graphics cards, based on the GPGPU tool cuda\cite{28}, and also a hardware solution for ray tracing by the company Caustic Graphics. Both systems claim to be able to display interactive real-time ray tracing results.} or other graphics companies, like it is often stated in the last months, that ray tracing will be the replacement of rasterization. For that the speed advantage of rasterization is just too big. However, by the use of hybrid approaches, some computationally intensive effects could be bound into the content pipeline a lot simpler and some others would even been just made possible by that. Current real-time effects, which could profit of this, would be (soft-) shadow algorithms or real-time global illumination algorithms that use ray-tracing-based photon mapping to determine the indirect light radiation. Within this semester thesis for the advanced seminar, the two topics of new ray tracing APIs and optimization of an existing GPU Ray Tracer, called RayGLSL will be focused. Furthermore, two more responses to Intels plans have arrived on the market. NVIDIA\textsc{'}s NVIRT API\cite{29} for NVIDIA graphics cards, based on the GPGPU\footnote{http://en.wikipedia.org/wiki/Rasterization} tool cuda\cite{28}, and also a hardware solution for ray tracing by the company Caustic Graphics. Both systems claim to be able to display interactive real-time ray tracing results.
1 Introduction

This semester thesis was created in the context of the Master of Science advanced seminar, held at the Munich University of Applied Sciences [31]. The main focus was laid upon upcoming real-time ray tracing APIs and hardware as well as optimizing an existing shader implementation of a ray tracer.

1.1 Overview

There is a large difference between ray tracing and rasterization. While rasterization was born to display geometry very effective and fast on early hardware, ray tracing was created to model a very computational intensive image of physically based light interaction processes. But the speed advantage of rasterization was bought at the expense of robustness and flexibility. Today there is a current and increasing need for newer and bigger effects to get consumers to buy the last bleeding edge titles. But the creation of these effects is a time consuming process. In current games it is mostly the largest investment and the content team is in most cases more than two third of the manpower of a game developer team. Even more, most of the effects are not very robust, because they are only simplifications of the physical processes in the real world. An example would be bump mapping, which is only correct if one does not look at a surface from very small angles. In ray tracing one expects a very clear shift to a more simplistic pipeline, because effects like reflection or refraction are delivered within the traversal engine. However, this does not mean that there aren’t any faked effects in ray tracing.

1.2 Motivation

Due to the immense increasing power of GPUs, it is possible, to parallelize streamable algorithms massively. The current GPU hardware exercises this option and uses a huge amount of transistors, to have a high SIMD \(^1\) bandwidth to handle the parallel data. Activities such as out-of-order-execution \(^2\) or branch prediction \(^3\) are not necessary on the GPU. A lot of transistors can be used to hide the latency between stream processors and memory accesses. With new programmable functions of the graphics pipeline, the GPUs became more and more attractive for so called general purpose computations on graphics processing units (GPGPU). That made the graphics card attractive for a lot of other areas than graphics. And even more this enormous computational power is

---

\(^1\) Single Instruction Multiple Data (SIMD)


\(^3\) [http://en.wikipedia.org/wiki/Branch_prediction](http://en.wikipedia.org/wiki/Branch_prediction)
not only applicable to rasterization these days, but also for other graphical algorithms. That is why the killer app for GPGPU is named graphics, jestingly. These new fields of graphics programming are maintained and enhanced by the market leaders in the GPU segment as well as other small groups. NVIDIA developed the compute unified device architecture (CUDA) API for this purpose. It is based on the language C and only works with NVIDIA GPUs. ATI has developed a similar but assembler based technology called close to metal (CTM) for its own series of GPUs. Another, but independent, GPGPU computing language is BrookGPU. Currently, there are two new ray tracing tools that will be launched very soon which will be examined in this paper.

On the one hand NVIDIA prepares its response for Intel’s plan, to define ray tracing the new killer app for their new processor. Its a new API, which is supposed to demonstrate that GPUs are also very well adapted for this task. This new API is called NVIDIA interactive ray tracing (NVIRT) and builds on top of cuda. At SIGGRAPH08 they were able to show a demo of a GPU ray tracer. In the context of this thesis, the Munich University of Applied Sciences was able to participate in an alpha testing program of the NVIDIA nvirt SDK. This work should give an overview of the API and the SDK. On the other hand, the desired and highly acclaimed hardware solution for ray tracing by Caustic Graphics is the second focus on new APIs in the ray tracing sector. The approach here is to build a card similar to the graphics card architecture but which is specialized on tracing rays. The approach is briefly presented and its pros and cons will be discussed. At the time of writing of this article the card was still in beta stadium which means that there is not that much of information given for this product. In addition to the new Intel hardware there will also be new solutions for parallelized computing, OpenCL and also the upcoming Compute Shaders which are part of DirectX 11. The first is a very general approach for connecting CPUs and GPUs in order to make better use of the immense computational power of todays machines. The second is a generalized GPGPU API bound to the DirectX API. But both are out of scope of this thesis.
2 Ray Tracing

2.1 The Algorithm

Initial efforts in the fields of ray tracing have been made in 1963 by Arthur Appel, Robert Goldstein and Roger Nagel at the University of Maryland. The algorithm traces rays starting at the eye point, through a viewport into the scene. Figure 2.1 demonstrates the algorithm visually.

![Ray Tracing Algorithm Diagram](image)

Figure 2.1: The ray tracing algorithm

The contribution to the light calculation of a ray to the according pixel is calculated at the nearest hit point between a ray and the scene geometry. This algorithm is recursive. On every hit point, depending on the material it is possible that there will be a reflection or a refraction ray created. These rays have to be traced further until a predetermined recursion depth is reached. Then all the results are returned back to calculate the final fragment. There is only little overhead required to combine the results. In addition to the reflection and refraction calculations at the hit point, shadow rays can also be spawned. These rays are sent towards each light source. If they hit any other object between the hit point and the light, the hit point lies in shadow, and the light arriving there will be attenuated. This type of ray tracing refers to so called local illumination, which does not take any ambient light into account (besides a constant term) in the calculation for diffuse materials, e.g. a diffuse white wall, located next to a red wall will not receive any red light that bounces off the red wall. Likewise, caustics (refractions and bundling
of light at curved transparent surfaces) are not possible in this way. For receiving these kind of effects more extensive procedures such as Path Tracing or Radiosity approaches are needed. James Kajiya and David Immel combined the illumination of a point in space into a four dimensional integral, the so called “Rendering Equation” \[19\].

\[
L_o(x, \omega, \lambda, t) = L_e(x, \omega, \lambda, t) + \int_{\Omega} f_r(x, \omega', \omega, \lambda, t) L_i(x, \omega', \lambda, t)(-\omega' \cdot n) d\omega'
\] (2.1)

It(2.1) states that the total amount of light \(L_o\), leaving a surface hit point \(x\), can be determined by the amount of emitted light \(L_e\) and the reflected radiation \(f_r\), integrated over the hemisphere over \(x\). The term \(f_r(x, \omega', \omega, \lambda, t)\) stands for a bidirectional reflectance distribution function (BRDF\[^1\]) that defines the distribution of light at point \(x\), from outgoing direction \(\omega'\) to incoming direction \(\omega\), at time \(t\) and wavelength \(\lambda\). \(L_i\) is light of wavelength \(\lambda\) coming inward towards \(x\) attenuated by the cosine between \((-\omega' \cdot n)\) due to incident angle. It is not possible to solve this equation for all cases. In fact it is only possible in very simple cases. But there are various numerical approaches for approximating the equation. For ray tracing in interactive real-time fields most of the time simple Whitted Ray Tracing will be used, as it only considers specular reflections and refractions and neglects the diffuse term in the integral. In this way, indirect illumination is disregarded.

In addition to the approximation of this equation, one of the main problems is the traversal of the geometry. Intersection testing is very expensive, because it has to be calculated for each primitive, against each ray, and the number of rays can grow explosively. Early ray tracing algorithms have had only very simple geometry, which is very cheap to test, like spheres and planes. However, in current state of the art pipelines such simple geometries are not usable. They mostly depend upon triangles or even more complex curved surfaces. As the hit point calculation for these primitives is quite expensive, further algorithms were searched, to minimize the costs of geometry traversal.

### 2.2 Acceleration Structures

#### 2.2.1 Uniform Grids

Uniform grids represent the simplest form of acceleration structures. They divide the scene space into equally sized cubes. These are arranged in a regular manner (the grid) and can be tested at a low budget. By this arrangement the calculation of hit points in this structure is very simple and straightforward. But due to the non adaptive nature of this approach, a so called teapot in a stadium problem arises. This indicates that scenes with very differently scaled geometry will have huge problems while tracing. So a teapot is very small compared to a stadium. But the geometry of both is equally complex. So due to the huge size of the stadium, the teapot will fall into only very few grid cubes. This means that if a ray hits the box of the teapot, it still has to traverse all

\[^1\]http://en.wikipedia.org/wiki/BRDF
of the primitives (triangles) the teapot consists of. That way the advantage of hiding the expensive intersection calculations is gone. Because of that, further methods of dividing scene space have been invented, which take the density of scene objects into account and divide the acceleration structure adaptively.

2.2.2 BVHs

A first step to achieve effective structures that are traversable in a reasonable amount of time is to subdivide the scene space into simple to test subareas or sub objects. From this approach the so-called Bounding Volume Hierarchies are derived. The complex primitives are packed into these areas and they only have to be further investigated or tested if the bounding volume of the containing node has been intersected. Bounding volumes are represented by very simple to test primitives such as the former mentioned spheres or boxes. Because it is very efficient to calculate if a ray traverses through a Bounding Volume by testing it against the center and the radius of a sphere or the 6 planes that define a cube. Figure 2.2 (middle) shows a 2D example of a BVH that consist of spheres as nodes, which contain the scene geometry.

Compared to KD-Trees (explained in the next paragraph), BVHs are very simple and inexpensive in construction. BVHs have the ability to be able to recreate parts of their structure incrementally without having to reconstruct the whole architecture. This attribute makes them very attractive for dynamic scenes because here the acceleration structure has to be recreated with every geometry update. In general, the resulting traversal speed is not as fast as the one of KD-trees. This follows from the fact, that BVH primitives can overlap and even if an intersection is found in one bounding volume, the ray has to be tested against all the primitives in the other bounding volume, which overlaps the first one. Also, the redistribution of bounding volumes where the containing primitives are drifting apart in dynamic scenes is non-trivial and can lead to huge, very inefficient bounding volumes.

2.2.3 KD-Trees

KD-Trees follow another approach. The KD stands for k-dimensional. For ray tracing very often 3 dimensions are used. Figure 2.2 shows an example of a 2D KD-Tree on the right hand. KD-Trees follow the approach to subdivide the scene space as cost-efficient as possible. For that, a cost model mostly referred to as surface area heuristic (SAH) is set up, such as the one described in [32]. Under a good cost-model approximation the scene space can now be divided optimally. It is also possible to define additional criteria for building a KD-tree. The creation of empty spaces that can be traversed extremely fast could be taken into consideration for choosing the split planes. The construction of KD-trees is very expensive and not unique since it is not trivial to choose the best of the three splitting planes. The leaves of a KD-tree solely contain the pointers to the primitives. It can also occur that a primitive is present in more than one leaf if the split planes divide it. The KD-tree excludes by definition that two of the bounding intervals overlap. But due to the complexity of building the structure of a KD-tree, they are more
suitable for accelerating rigid (static) structures. However, for dynamic scenes, hybrid approaches are conceivable. For example, the ground structure is a BVH, which holds KD-trees in its leaves for the rigid atomic parts that are not moveable or deformable.

2.3 Caustic Graphics

2.3.1 CausticRT Overview

In 2010 the company Caustic Graphics will deliver its first product to the market, which is currently in beta status. It is a hardware card which calculates 3D scenes similar to a graphics card, but by using ray tracing, instead of rasterization. The product of the company is the ray tracing solution CausticRT. It consists of the video card CausticOne and the associated API CausticGL. The company names high quality image calculation for movies or equal as well as real-time applications, primarily games, but also visualization tools as potential field of application. The graphics card will not be replaced, it is still needed, but the ray tracing is done on the Caustic card. It is therefore a kind of coprocessor for ray tracing. Even though CausticOne cards are in discussion for render farms, the company targets another field in high quality graphics. Until now, high quality images are created in development environments, which, because of speed issues, shade the geometry by rasterizing it, while the developer works on the scene. As these scenes are very often extremely complex, it is not possible to display the lighting calculations with a ray tracer in real time. Later, however, production renderers like Mental Ray, VRay, Final Render or Brazil are used to do the shading for the scene with computationally intensive methods like Ray Tracing, Radiosity or Photon Mapping. This means, that the final images differ hugely from the images, worked on at development time. At this point Caustic Graphics wants to present a solution, which is able to display a real-time capable, dynamic view of huge scenes and the associated shading at interactive frame rates. With their first evolutionary stage of their card Caustic Graphics promises a 20 times speed up, compared to other current interactive real time ray tracing procedures. What current procedure and hardware they relate to, has not been mentioned yet by them. The first product, which has been ported to use Caustic’s API, is the production renderer Brazil. The renderer was developed by Splutterfish and acquired by Caustic. Brazil is a well-known renderer, which can be integrated to 3ds max or Rhino3D and produces high quality images as output.
Caustic Graphics hopes to present a platform using CausticRT with a fully implemented production renderer. The implementation to CausticRT should happen within the next release of the renderer.

### 2.3.2 API, Design and Hardware

Figure 2.3 illustrates an overview of the CausticRT platform. It shows that the Rendering API builds upon OpenGL ES 2.0 and GLSL. This shall give developers a well known API as a basis which enables a steep learning curve. Furthermore, it should offer the possibility to implement the Caustic approach into existing OpenGL Applications.

As figure 2.4 shows, the card will be connected to the system via the PCIe x4 BUS. The next evolutionary stage of the card, the CausticTwo, will be connected to a faster BUS. The pipeline is built in FPGA hardware, which is connected to 4 GB of GDDR3 RAM to be able to display bigger scenes. Until now there has not been published anything about a virtual, or streaming technology, to be able to do memory paging for huge scenes. According to the US patent application 20090128562 CausticRT implements an algorithm which gathers the rays internally in a BVH like structure. This structure is called Geometry Acceleration Data (GAD). It gathers cells until they are full or reach predefined waiting criteria, and then sends them to the stream processors for intersection testing. By this technique they try to minimize the enormous incoherence after the ray
casting phase. These BVH like structures have to be updated on the CPU, so there is no direct support for dynamic scenes onboard.

Figure 2.4: The CausticOne card

2.3.3 Possible Drawbacks

• Only IEEE single-precision support on the Caustic card. Both NVIDIA and AMD graphics card support double-precision. Intel will support it on Larabee. This could be too inaccurate for the high quality image field.

• CausticOne is rather a beta prototype. The real prototype with ASICs on board will be the CausticTwo and will be released next year. Until then the cards supporting NVIRT will also have made a huge leap forward. Also Larabee will be released and will have a decent community by then.

• Until now Caustic Graphics has not made any statements about methods for virtualizing memory to support large scenes. This will be essential in production renderer scenes.

• The price for one CausticRT SDK is currently 4000$. That includes one card and the API usage. For another 2500$ one will get software and firmware updates, access to the developer forum and one year of support. In addition to this, the system requires a powerful GPU for shading and a strong CPU for updating the accelerating structures. It may be questionable if one cannot get a more powerful CPU or GPU system for the same amount of money.

• Possible bandwidth problems. The ray intersections have to be transferred via PCIe x4 BUS to the graphics card for shading the geometry. Maybe a current GPU System will be far more superior if tracing is done on card, and the data can be held on card for shading.


2.3.4 Prospect

Figure 2.5 shows some high quality images rendered with the CausticRT SDK. The scene includes two cars and a fountain filled with water. The cars show reflection effects and the water in the fountain has an additional refractive material. Notable are also the soft shadows displayed in the scene. The scene is not dynamic and consists of more than 5 million triangles. The picture was computed at a resolution of 720p. The setup includes a TrueHD pipeline with trilinear filtered textures and anti aliasing. There are 16 ambient occlusion rays and 16 area shadow rays per pixel at 8 samples. For glossy effects on the rims additional ambient occlusion samples are used. The render times for the images are (from left to right) 3, 4.8, 5 and 6.9 seconds. The images can be edited within the CausticOne development environment at 3 to 5 seconds per frame. Time will tell if this ray tracing method will be accepted. Similar approaches have been tried with SaarCOR or the Ray Processing Unit (RPU), the successor of SaarCOR and have nearly disappeared. It will be a difficult act of balance for the people at Caustic Graphics to find buyers for a product that very few people have developed for. On the other hand it is difficult finding developers for a system, which is not very widespread. From the viewpoint of a graphics artist a solution for on the fly-editing of ray traced scenes in a ray traced development environment will be welcome with open arms. It remains to be seen if the Caustic team can hold what they promise.

2.4 NVIDIA’s NVIRT

NVRT stands for NVIDIA Interactive Ray Tracing and is a new API that makes ray tracing on current GPUs simpler to implement. It is not a renderer, its purpose is also for general ray tracing tasks like AI queries or collision detection.

2.4.1 About NVIRT

Of course there are already applications implementing ray tracing on current GPUs using shaders or the cuda API. But all of them have disadvantages, which if NVIDIA has its way will be corrected with NVIRT. Generally, the usage of rasterization is always the fastest option for tracing primary rays (Ray casting). Because these have a high coherence and can thus effectively use the hardware of the standard graphics pipeline. What might come in a little unexpected is the fact, that the Rasterizer is quite handy for ray casting. It

\[http://www.caustic.com/gallery_images.php\]
allows to effectively evaluating the depth values for each fragment of the viewport, which is equivalent to finding the hit points of the primary rays and the best part is, it happens in hardware. A disadvantage of the cuda API at this point is, that it had the need to copy the primary ray data from a rendering API Context like OpenGL or DirectX into its own Context. This was not possible to be implemented very efficient in the graphics card drivers. This does not apply to a shader implementation of a ray tracer. But on the other hand the implementation itself is more difficult and the maintainability is not as successive. There is no emulation mode or a debugger like in cuda to investigate runtime errors. Also the architectures of ray tracers, developed with shaders arent very intuitive, as they are using the rasterization pipeline for a ray tracing purpose. So textures are converted to arrays, pointers become indices, the viewport must fit the fragment indices of a quad that holds the results, in height and width. These and other considerations led to cuda and other GPGPU APIs, prior to ray tracing. They allow direct communication with the stream processors and their cache, or the GPU memory. Shaders got replaced by the kernel, which works similar to the shader, on fragments, which are the atomic thread unit.

2.4.2 NVIRT Overview

NVIRT is planned to be released in spring 2009. It will be part of the NVIDIA Scene Graph SDK (NVSG SDK) but without any dependencies to NVSG. It is meant to be a low level, high performance ray tracing API. Until now NVIRT only runs on NVIDIA Quadro FX cards. NVIDIA already showed a GPU ray tracer at the SIGGRAPH08, which was based on a shader implementation and had no NVIRT support. The according slide showed some ideas that were included in NVIRT. NVIRT, as a generalized approach to ray tracing, does not produce the same speed yet, but according to NVIDIA the same speed level should be reached in the beta phase. Figure 2.6 shows two scenes captured by the ray tracer.

The scene contains 2 million triangles and was rendered with a resolution of 1920x1080.
Frame rates of up to 30 frames per second were achieved. The geometry is dynamic, including a moveable light source and a recursion depth of five bounces per reflection / refraction. The underlying hardware was a Quadro Plex System with four Quadro FX 5800 cards, each with 240 stream processors and 4 GB of RAM. Whereby it must be mentioned, that the memory does not summarize. Because on the one hand each card needs the complete scene on board to traverse it, but on the other hand each card can render a different section of the screen. This demo showed that todays graphics cards can also assist as a ray tracing coprocessor due to their GPGPU ability. Even if the test system for this demo cannot be compared to todays usual consumer PC, in a few years, with the current development of GPUs it can be seen as an alternative to current ray tracing CPU systems.

The NVIRT API is based upon cuda. It uses the virtual assembly language called ptx of the cuda compiler nvcc to compile its programs. NVIRT programs can also be programmed with a C wrapper that is built in to the compiler. On the host (CPU) side NVIRT can even be programmed object oriented in the C++ language. Rays or axis aligned bounding boxes are additionally included as types into the cuda API. Multiple cards in a system can be used concurrently for computations. But it is not as low-level as the SLI interface, the graphics devices can be fed with programs and data directly via device indices available in the cuda context.

2.4.3 The NVIRT API

The ray tracing pipeline has programmable interfaces, which define the program flow. They work on atomic fragments or define them. Thus the flow is defined as if one would program for a single ray.

2.4.3.1 The Elements of NVIRT

Figure 2.7 is an overview of all the basic elements NVIRT brings with it. Central element of each application accelerated by NVIRT is the Context.

Figure 2.8 demonstrates the workflow of a GPU based ray tracer. It should be noted, that NVIDIA itself recommends using the Rasterization pipeline for tracing primary rays (ray casting) because it is faster for this task. (David Luebke and Steven Parker: Tracing eye rays is uninteresting - rasterization wins, use it. Slide 44, [27])

The programs presented on the right hand side of figure 2.8 are all candidates for a parallel execution on the stream processors of the GPU. The Context of a NVIRT application holds the ray generation programs (e.g. camera), exception programs (e.g. predefined colors for incorrectly calculated fragments), miss programs (e.g. similar to the background color in the standard rendering pipeline) and user variables. The Context is not restricted to use only one ray type. In later stages of the pipeline different ray types defined in the Context, e.g. radiance rays and shadow rays can be used. All the programs are defined in .cu files so they can be translated to ptx assembly. It is possible to define own variables in these programs set by the host via the context.

3http://hhohardw are.com/News/NVIDIA-Shows-Interactive-Ray-TRacing-on-GPUs/
Figure 2.7: The elements of the NVIRT API[29]

Figure 2.8: Der Ablauf in die parallelisierbaren Stufen eines GPU Ray Tracers[27]
Furthermore it is also possible to access semantic variables like the ray index from within programs. The generation of camera rays depending on the 2D viewport coordinate would be a candidate for such a program. The tracing of rays is done again in such programs defined as intersection definition programs for each primitive. So every used geometry primitive (triangles, spheres, boxes) will define a BoundingBoxProgram and an IntersectionProgram. In this way, even primitives with curved surfaces are possible.

The geometry objects of the scene are contained in so called GeometryInstances linking several materials to a geometry object. A Material in turn is a container that holds the programs needed for shading the surface of a primitive. The AnyHitProgram and the ClosestHitProgram are linked to a surface at this stage. AnyHitPrograms are needed by the traversal pipeline to define the further behavior at hit points. So for example in an AnyHitProgram it is possible to ignore hit points for shadow ray creation if the material is transparent at the hit point. An Example for an AnyHitProgram is explained in the Whitted Example Program flow. ClosestHitPrograms define the lighting calculations for shading a hit point on the surface of the underlying geometry. The GeometryInstances in turn are ordered in GeometryGroups. They include the according Transforms and Selectors. Similar to a scene graph node, they are able to manage the movement and rotation of a node relative to its parent node. Selectors describe various choices of the linked geometry. They can implement LOD practices for choosing the right mesh for the intersection test. Further elements of the Context are the so called Buffers. They are in a figurative sense the arrays that are used by the programs running on the stream processors. In addition to formats like RGB (float3), RGBA (float4) or Luminance (float) and other standard formats NVIRT also allows the usage of user defined formats for textures. This can be used to setup light buffers for the usage in programs. NVIRT buffers are also the interaction interface for the OpenGL API. Before using the buffers the usage permissions have to be set. Possible values are InputBuffer, OutputBuffer as well as attaching both flags to a buffer. Input means that only the host is allowed to fill a buffer with values and the device is only capable of reading them. The same applies vice versa for OutputBuffer. Internally the buffers will be extremely optimized, so it may not be a good idea to declare an often used buffer both, input and output. A render target always has to be an output buffer. This way it can be mapped into the OpenGL context.

A final but very important component of the Context and the scene is the acceleration structure object. It is not mandatory to use a particular one or to use one at all. They can be user defined as well. NVIRT also provides several predefined methods for acceleration structures. Thus, besides a KD-Tree also a BVH can be laid upon the scene geometry. The BVH program can even select from different acceleration levels. A distinction is made here in the building speed of the structure. It determines how well the acceleration structure will fit the underlying geometry and how effective the traversal will be. If geometry is getting deformed or translated, the context will notice and mark it as dirty. Then the acceleration structure will start to rebuild itself. The previously described GeometryGroups are further aggregated into Groups, which relate an acceleration structure with different GeometryGroups that are watched by the structure. If the Context is defined and compiled, its Trace Method, which starts the ray tracing in 1D, 2D or 3D can be called. Figure [2.9] illustrates this connection inside the NVIRT API.
2.4.3.2 Mode of Operation of an NVIRT Application

A scene displaying Whitted Style Ray Tracing, as shown in Figure 2.10, is used as an example. Whitted Ray tracing generates a ray from the eye point through every fragment of the viewport and sends it into the scene. These rays are traced backwards into the scene. At the nearest hit point new rays are generated and sent into direction of every light source. These are the so called shadow rays. If there is no other geometry between hit point and light source the point is not shadowed and thus shaded normally. Otherwise this fragment will get an attenuation factor. If the hit primitive has a reflective or transparent material, further rays are generated. Reflection and refraction rays are sent from the hit point into the scene. They also contribute to the illumination of the fragment. The ray generation process is stopped if the contribution is under a certain amount or if a predefined recursion depth is reached. The process of this Method is illustrated in figure 2.11.

The Setup for the Context stipulates that there is an entry point for the ray generation, the camera. In this case, a simple pinhole camera is used calculating the rays via their ray index, according to the viewport fragment they pass and the eye point. Furthermore, two types of rays will be investigated, radiance rays and shadow rays. There are 3 kinds of primitives in the scene: spheres, shells and planes. As stated before, for every primitive there has to be an own IntersectionProgram and BoundingBoxProgram. They will be attached to the Context. Also every Material in the scene has to describe the behavior of its surface if a ray hits an object with this Material. This is defined in the AnyHit- and the
Figure 2.10: One of the first scenes generated using ray tracing. This one is created using nvirt

Figure 2.11: Simplified workflow of the white stile ray tracing application in nvirt
ClosestHitProgram. The Material for the sphere in the background will appear metallic which means that in case of an AnyHit only the nearest hit point will be traced. In the ClosestHitProgram the point is phong shaded and a reflection ray is generated and traced. This is different for the glass Material. Here the intersection with the object is ignored at first. It starts tracing the reflected and refracted rays to see if they hit any opaque object. Based on this information the lighting calculations for this hit point can be calculated. The simplest Material in the scene is the checkerboard pattern, which is just evaluated via the texture coordinates of the hit point. Since the material is diffuse, there are no further rays generated. Figure 2.12 illustrates the object hierarchies in this scene. Apart from the ray generation program, the pinhole camera, there is also a geometry group attached to the Context. There is no acceleration structure attached in this application. If so, it would sit on the same level as the GeometryGroup and both would be contained in a Group object. The GeometryGroup holds the GeometryInstances, which link a Material to the according Geometry.

2.5 RayGLSL

RayGLSL is a ray tracer and was developed by the company LFK [10] as part of a diploma thesis. It is based on a shader approach and was evolved using OpenGL and GLSL. It was placed in respect to a second ray tracer, developed at the same time, which was called cuRT. It was developed as part of a master thesis and was also a GPU approach. But the
API used for creating it was cuda. The comparison should give a clue about the speed behavior of GPU ray tracers for future developments. In the end RayGLSL established itself. It demonstrated a definite speed advantage, which can be traced back to the effective tracing of primary rays by the rasterization stage. Besides the speed advantage, a GLSL approach is more independent cuRT will only run on NVIDIA hardware.

RayGLSL builds upon OpenGL. So it is suitable for any graphics card architecture with few adjustments (e.g. extensions for special texture formats). The ray tracer is splitted between GPU and CPU functionality. On the CPU, the static and also dynamic acceleration structures are created and maintained, if they have changed and need to be updated. The GPU application however is divided into two stages. As stated before, the rasterization pipeline is very efficient at tracing primary rays. So it was taken for task one. Following this stage shadows and, if present, reflection rays of a scene will be traced on the GPU. Special geometry data like vertices and normals are held in textures and can be accessed by a special indexing system. The chosen acceleration structure is a KD-Tree. It creates an optimal traversal structure for static and dynamic scene geometry as described in section 2.2.3. All though static geometry and dynamic geometry among themselves are held separated in different trees. The chosen optimization is a so called restart strategy for KD-Trees, introduced in [9]. A KD-Tree itself is represented as a texture on the GPU and also uses a special index structure to traverse its leaves and nodes. The creation strategy for the KD-Tree is a surface area heuristic (SAH). For the lighting calculations, there is an extra stage at the end. It calculates the fragment color from the primary, shadow, reflection rays and the material properties of the hit point. It also contains the texture mapping.

2.5.1 RayGLSL and SLI

As part of this thesis an overview of using the Scalable Link Interface (SLI) with a GPGPU tool like a ray tracer were meant to be assembled. As SLI is linked very deeply inside the graphics card driver it is not that simple and precise to work with as with a GPGPU tool like cuda. Cards used for SLI do not have to be by the same manufacturer, but they have to have the same chip series (8800, 8600 etc.) and the same model name (e.g. GT, GTS, GTX).

The three modes SLI works in and one additional mode:

- **Split Frame Rendering (SFR):** This mode splits the rendered screen into two areas. It analyzes the drawn geometry and varies the ratio between the two halves to balance the workload for both GPUs at 50/50. For GPGPU this approach wont work very well, as the only geometry is the quad, which the results are drawn to.

- **Alternate Frame Rendering (AFR):** In this mode each GPU renders one frame after another, so one GPU renders even frames and the other renders odd frames.

- **SLI Antialiasing:** This approach is not used to improve the frame rate, but to improve quality. Each GPU is used to render a different pattern of the image. Each fragment is shifted slightly and in the end the calculations are composited.
• NVIDIA \texttt{WGL_NV_gpu_affinity} extension: As none of the upper modes is really applicable to GPGPU programs, NVIDIA reacted with this extension for cards of the Quadro series. It introduces device contexts for directly managing cards and several functions to communicate this to the card.

One disadvantage of using SLI is the non-transparent programming of SLI for applications. It seems, SLI support has to be delivered via a config file, like done by NVIDIA for several games. SLI primarily aims at accelerating visual rendering with the rasterization pipeline and not GPGPU. It has lots of driver and bandwidth managing overhead and it doesn’t always preserve a speed upgrade even for games, which it was designed for.

2.5.2 Optimization Possibilities of RayGLSL

As shown in figure 2.13, RayGLSL spends 80% of its time traversing the KD-Tree. Within this stage lies great potential for the optimization of this application. For that reason this section primarily focuses on algorithms dealing with efficiently finding hit points of rays with primitives in acceleration structures.

2.5.2.1 Overview of the proposed Algorithms

The following sections will introduce current acceleration structures for GPU ray tracers. The presented algorithms use promising new approaches. To begin with, it was emphasized to find papers that are based on ray tracers implementing shader approaches. But in most cases the only shader stage used was the rasterization pipeline to generate primary rays. All of the approaches presented in this work are based on cuda as they were the newest and most promising. But the results should be portable to a shader approach.
The first article\cite{33} introducing an extended KD-Tree approach, which trades time for space. It uses a kind of adjacency pointers within the KD-Tree-structure to optimize traversal time. The second paper\cite{14} describes a method that uses a memory efficient BVH strategy. According to the authors it achieves same speeds as the just mentioned KD-Tree algorithm. Both approaches are designed and developed by the same authors. BVHs additionally promise a faster construction time, making them an attractive alternative for dynamic scenes. The BVH approach also allows the usage of huge (up to 12.7 million primitives) models on the GPU, as it is very memory conserving. Last, but not least, probably the most important paper\cite{2} discussed by this work will be addressed. This paper will be published at the High Performance Graphics Conference\cite{11} in New Orleans, USA. It includes a discussion of traversal strategies on current graphics hardware and its theoretical performance. Derived from this, problems of current algorithms are marked. Building on top of this knowledge, according to the authors, the currently fastest GPU ray tracer is presented.

2.5.2.2 Stackless KD-Tree Traversal for High Performance GPU Ray Tracing\cite{33}

As described earlier, RayGLSL uses a KD-tree approach with a SAH cost model as acceleration structure for traversing the geometry\cite{8}. Additionally a KD-restart is used, which considers certain aspects of the GPU architecture and optimizes them. All of the KD-tree algorithms on GPUs are stackless, because they are a lot faster. Stacks on GPUs produce a lot of kernel switches, which would introduce high overhead and huge memory bandwidth requirements for storing intermediate results. All of these stackless algorithms run on current hardware and have decent ray throughput numbers. But they are still behind CPU implementations like the packet ray tracer by Wald et al.\cite{36}. The article by Popov et al. \cite{33} describes an extended method of the \textit{kd-restart} algorithm. They add so called ropes to every node and leaf of the KD-Tree, shown in figure \ref{fig:2.14}. These ropes are a kind of pointer, which addresses a neighbor for each of the six faces of a leaf. So, every face gets an adjacent leaf assigned, or, if there are multiple neighboring leaves, the smallest node containing these adjacent leaves of a face is linked by the rope. Border faces of the tree, which do not have any neighbors are linked to a special nil node. This algorithm can easily be set on top of existing KD-Tree building implemenations as a post processing step. It has a complexity of O(N). This technique increases the node size by 24 Byte, but the reduction of traversal time and cost amortizes this disadvantage of fetching more memory per node.

First an approach is introduced, that traces a single ray per thread through the tree. This method tries to assign the next traversed leaf to a ray that has not found any intersection in the current leaf. For that purpose the exit point of the current leaf is determined, and the according rope is fetched. From this the next leaf which has to be traversed can be found. If it is not a leaf, the sub tree has to be traversed, until the adjacent leaf is found. If the ray hits the nil node it terminates. Building on top of this, the actual algorithm is presented. It describes a way to send packets of rays down the pipeline. The algorithm builds upon the idea to let one packet work on one node. On a GPU one packet is a block of threads, which will not change multi processor. The packet
will exist until all the 32 rays have terminated. So, to get more coherent memory accesses, the algorithm decides which child node (left or right) has to be traversed first. This is decided by a criteria, illustrated in figure 2.13. This criterion also indicates whether the current packet is coherent. The ray itself holds data, which determines the current node for this ray. So called *active rays* are rays whose current node is the currently traversed node of the packet. This criterion is evaluated until a leaf is reached, where the rays are intersected or linked via a rope to a new node. If all *active rays* have terminated, the algorithm terminates and returns to the other not yet processed node. If none of the active rays have a node to work on, the packet terminates.

Compared to the kd-restart algorithm the single ray traversal with entry point optimization approach saves up to $5/6$ of the down traversal time. Furthermore, the down traversal of *kd-restart/kd-backtrack* is more expensive as both algorithms need to intersect the ray with the split plane, in contrast to a simple point location query, performed by the stackless traversal algorithm. It is shown that the memory usage of a KD-Tree with ropes does not surpass a factor of 4 times the memory used by a standard KD-Tree implementation. The examples in the paper used a factor of around 3 times more memory.

### 2.5.2.3 Stackless KD-Tree Traversal for High Performance GPU Ray Tracing

RayGLSL divides the scene into a static and a dynamic part. Furthermore, the dynamic part is again subdivided into atomic rigid units which form their own KD-Trees. Each of these KD-Trees has to be traversed for the computation of one frame. Subdividing the dynamic parts into atomic rigid units has the advantage of not needing to rebuild the KD-Trees. However a disadvantage lies in the fact, that all these KD-Trees have to be tested against each ray in the beginning. The last section introduced a way to optimize KD-Trees. This section shall present a new approach that uses BVHs for acceleration. This approach also makes it possible to bring huge scenes onto the graphics card, due to reduced memory usage. The authors present the power plant scene, which consists of 12.7 million triangles. It is rendered with the proposed approach at a resolution of 1024 x
1024, achieving up to 3 frames per second. The scene is shaded and makes use of shadow rays. They claim that this approach is also very efficient for incoherent secondary rays. The method is based on BVHs created by a SAH related to the ones used in KD-Trees.

The presented algorithm also makes use of ray packeting to hold the workload up. The rays in a packet always traverse a node at once. For a G80 this means 32 rays per packet, because that’s the atomic unit of a block (32 threads per block on a MP). Further the authors show, that current graphics hardware is very close to so called concurrent read concurrent write PRAM machines, which the authors use as programming model, that their algorithm builds upon. As opposed to the KD-Tree rope approach, they use a small stack for the BVHs. It is located in the shared memory of the multi processor.

A packet is now traversed through the scene and hits the first node. If it is a node, each ray decides for itself, which half to traverse further. For the left half it writes a 1, for the right half it writes a -1 into the shared memory. By using a PRAM sum reduction the method evaluates which half of the node has more rays (sum >= 1 equals left, otherwise right) and is traversed first. The complexity of a PRAM reduction is $O(\log N)$. This means that the sum for 32 threads is computed in 5 steps. The half that is not visited first is pushed on the shared memory stack for later use. If the stack is empty after the traversal of a node, the packet terminates. For the used intersection tests they refer to [21].

A second part of such dynamic scenes is the construction of the BVH structure. It has to happen with every geometry update, which can occur several times per second. The ground structure used for the BVH is a binary tree with axis aligned bounding boxes (AABBs) as container. For construction a technique called binning is used to approximate the SAH equation used. The construction still takes place on the CPU, but they announced a GPU version for the future. The cost function and the optimized approximation can be read in depth in the proposed paper, as they are out of the scope.
They show that streamed binning SAH BVH construction is a lot faster than SAH KD-Tree construction, even though it is more computational demanding. But because a BVH does not split primitives, fewer nodes need to be created and BVH trees often have less tree levels than KD-Trees. The created BVH structures were 3 to 4 times smaller than a standard KD-Tree, not to speak of a KD-Tree with ropes. For primary rays the BVH structure even was slightly faster in some cases. This can be due to the fact, that the occupancy of the GPU was held at 63% compared to 33% for a KD-Tree approach.

2.5.2.4 Understanding the Efficiency of Ray Traversal on GPUs

In this approach the two authors try to investigate the two elemental operations of ray tracing: acceleration structure traversal and primitive intersection by bringing them to a SIMT simulating machine. They try to compare the optimum simulated performance results to measured ones on real GPUs. While their focus is on NVIDIA GPUs, the results should be transferable to other GPU chips. With their work, they want to reveal current losses in performance of current GPU ray tracers. For that they developed a simulator which can estimate the upper bound for a given ray tracing kernel. They observed that well known methods are 1.5 to 2.5 times away from their optimum performance. Most of these gaps cannot be declared by memory bandwidth insufficiencies, but by previously unidentified inefficiencies in hardware work distribution. They developed a solution which nearly fills the gap from simulation to measurement. By providing their result for primary, ambient occlusion and diffuse interreflection rays, they show practical results of their approach.

The authors base their research on an abstract trace() method that holds both, the traversal and the intersection code. The problem in GPU ray tracing utilization is not limited by computation, memory bandwidth or something else. Even though the results may depend upon the scene setting, the situation over all is poorly understood for GPUs. They present different approaches for trace() implementations and compare real cuda kernel implementations with their simulator. Then theoretical possible values can be compared with the measurements. The identified main culprit is the hardware workload distribution. What causes the inefficiency is the discrepancy between rays in a packet that want to traverse and those that want to intersect. Because the code for those operations cannot be executed concurrently in threads. Their test setup consists of an optimized version of the BVH developed by Günther et al. [14] for traversal, and Woops unit triangle intersection test. The graphics card used for their measurements was a NVIDIA GTX285.

---

4the SIMT/SIMD is a superset of SIMD with execution divergence handling built into hardware. SIMT typically computes all control decisions, memory addresses, etc. separately on every (SIMD) lane [Lindholm et al. 2008]. The execution of trace() consists of an unpredictable sequence of node traversal and primitive intersection operations. This unpredictability can cause some penalties on CPUs too, but on wide SIMD/SIMT machines it is a major cause of inefficiency. For example, if a warp1 executes node traversal, all threads (in that warp) that want to perform primitive intersection are idle, and vice versa. We refer to the percentage of threads that perform computations as SIMD efficiency.
if-if trace():
while ray not terminated
    if node does not contain primitives
        traverse to the next node
    if node contains untested primitives
        perform a ray-primitive intersection test

Figure 2.17: The if-if trace() method

For the simulator they determine the operations, needed for traversal and intersection. This is done by evaluating the assembler code generated by the nvcc. Then they run the so called ray dump on their simulator whilst being able to choose their own scheduling method. The basic setup for the simulator is configured in a way that it is impossible for a GPU to be faster (infinitely fast memory access, SFU always filled). The simulator provides a hard upper bound for the performance. First they investigate packet traversal. As one would estimate for coherent primary rays, there is no performance hit. But they found a difference of 1.7 to 2.4 between simulated and measured data. On their way to find the answer they examine several different trace() methods, starting with per-ray traversal. First they show a so called while-while implementation of ray traversal.

while-while trace():
while ray not terminated
    while node does not contain primitives traverse to the next node
    while node contains untested primitives
        perform a ray-primitive intersection test

Figure 2.16: The while-while trace() method

Because of the less coherent memory accesses of per-ray traversal than that of packet traversal one would expect a larger difference between measured and simulated data for per-ray traversal. But the difference is not that big. Also very interesting is, that the theoretical upper bound performance of the if-if setup is 20% less effective in simulation than while-while. But the measured Mrays / second are equal to the while-while ones.

Further more interesting is that if-if, despite theoretically disadvantaged, can outperform the favored while-while. if-if has fewer exceptionally long-running warps and so the workload will not be disturbed that much. Another hint is that despite equally distributed warp-execution times, the slowest if-if warps are 30% faster. The authors claim that this must have something to do with the work load distribution. All NVIDIA cards have a special work distribution unit on board. It is specialized on homogeneous units of work, which is badly applicable to ray tracing. Because long running rays keep whole warps occupied.

To investigate this workload issue they write a special persistent kernel, which fetches work himself from memory, while keeping the MPs busy. For packet rays, they achieved a 1.5-2.2 times speed up and the performance is now within 10-20% of the theoretical
upper bound. They state that one cannot come very much closer to the optimum, because that would imply, among other things, optimal dual issue, complete absence of hardware resource conflicts, all memory access latencies to be hidden, and all of the > 20K concurrent threads to terminate exactly at the same time. They also found out, that memory bandwidth cannot significantly be the issue, because the more incoherent while-while has a similar speed up behavior as the coherent packet ray approach. If memory bandwidth was the problem, the more coherent ray packets should perform better, due to better use of the texture cache.

They further illustrate a so called speculative trace() method. As the tracing is divided into two stages, a warp can only execute one of the stages at once: structure traversal or primitive intersection. It makes use of the fact that rays in a warp which already have found a leaf while traversal, will have to wait for rays, which are still traversing. As they have to wait anyway, it may be beneficial to let them participate in the traversal of the still executed rays. There is no memory overhead, as the node data will be fetched anyway. It should improve the performance whenever the speed of the memory subsystem is not limiting the execution speed. They achieve a 5% speed up for very SIMD friendly primary rays, and up to 10% for ambient occlusion rays. Diffuse rays were not able to speed up, which is a first sign for being memory limited at incoherent memory fetches. They introduce additional ways to improve the simulated theoretical SIMD efficiency for future architectures, like replacing terminated rays with new rays. For this approach, strategies using (on GPUs not yet available) warp-wide instructions are also presented.

2.5.3 Implementing Billboards and Alpha Textures into RayGLSL

Billboards are quads which are aligned to the viewport of the camera. They are always parallel to the viewport and thus they always show the same face to the viewer. They were developed amongst other points to display outdoor vegetation effectively. Until recently, it was not possible to display complex geometry like trees very fast. But also this complexity makes it very hard for the human perceptual system to tell if a far away tree is real geometry or if it is only drawn to plane. Billboards make use of this effect. Until now, they are not implemented into the ray tracer. All the geometry in the scene is represented as triangles and sent to the GPU in a texture. As billboards also consist of two triangles, no specialization has to be done here. Only the orientation matrix has to be fetched from the viewing system to arrange them facing towards the camera. Further a new Material has to be created that uses a special texture format. RayGLSL only uses RGB texture formats for materials. An additional channel is needed that stores the alpha value for a texture. If this happens to be bandwidth problem also a color value could determined in the material that defines transparent areas, so that only alpha testing is supported. With this material, transparent areas of a billboard can be defined where rays can pass through, without surface shading at the intersection point. These rays keep their direction and achieve the hit point (with a little delta in direction of the ray, so re intersection with the billboard is prohibited) as new origin. These rays can be fed into the next traversal stage of the traverse. The billboard is no more relevant for the evaluation of the fragment color. It should be noted, that billboards should not
generate any reflected rays, because due to their flat nature normals would not seem very adapted to the faked geometry. This is already implemented for diffuse material and can be extended for the transmissive rays. In the end it should only a material that has to be created, that is able to handle alpha colors or alpha values in RGBA textures.

2.5.4 Prospect and Discussion of the Proposed Optimization Techniques

This section showed, that new GPU approaches are very competitive compared to CPU ray tracers. Comparing them is not that easy, as they don’t use the same test setup. The kd-Tree implementation in section 2.2.3 achieves 12M rays/s for the conference scene. The setup for this is 512x512 primary rays, supporting area lights, reflections and refractions. All though they found a discrepancy in rays using too much cycles which they will investigate further. For the BVH paper the authors claim to reach the same speeds as in the KD-Tree paper for a scene setup of 1024x1024 primary rays with additional shadow rays. In their paper they show two tables comparing the BVH approach to their KD-Tree method. It shows that their BHV approach is even slightly faster for most scenes. In a second table, they also compare the memory usage of both, which illustrates the advantage of the BVH approach. Both approaches used a NVIDIA GeForce 8800 GTX for computation. The last paper by Aila and Laine achieves even for randomly shuffled global illumination rays an average performance of 20 - 40M rays/s and even much more for coherent rays. This is really impressive, as it shows that incoherence is not a big issue for GPU ray tracer as thought before. More results for primary / ambient occlusion and global illumination rays can be viewed in table 1 in their paper [2]. But for comparison with the first two approaches it has to be said, that their approach uses a NVIDIA GeForce GTX285, which has more than twice as much stream processors than a 8800 GTX. It is also difficult to compare these results to RayGLSL, because there is no M rays/s information in the thesis related to RayGLSL. It only relates pure frames per second to the resolution used. As there are shadow and reflections used in the scenes, it is not possible to infer from the frames per second to any M rays/s numbers. All of the three approaches shown here have potential to achieve a decent speed up for a GPU ray tracer. Also all of them use cuda, and maybe in the future the inter op between cuda and OpenGL will get better and also the ray casting disadvantage will disappear.
3 Conclusions

3.1 Summary

The Chapter about ray tracing APIs shows, that this field is growing permanently. Several new technologies are still in the making and will hit the market next year. Caustic Graphics will have a hard time competing with the big companies in graphics card business like AMD, NVIDIA and also Intel. Maybe they have found a niche with the real time interaction of ray traced scenes that they can build upon. Like physics accelerator cards manufacturers already had to learn, the average gamer is not easily willing to spend money for additional cards. It will also be interesting to see the actual graphics hardware compete against more programmable approaches like Larabee. Next years paper will show, what can be done on a graphics card using the generalized NVIRT approach building fast ray tracing applications. Maybe an more independent approach like OpenCL can succeed? They claim to use the GPU and CPU concurrently as streamable processors, to get the last out of current machines. The three papers in the chapter about optimization strategies for ray tracing applications showed, that there is a lot of potential in new approaches on the GPU. The main conclusion which can be drawn, is that it is essential to keep the workload high on the processors. This can be achieved by using ray packets like the first two approaches used them. Also new structures that suit the GPU architecture better are presented. The rope approach shows that stacks for traversing KD-Trees on the GPU are easily outperformed by the used approach. The second paper also showed that huge scenes with a BVH acceleration structure that holds far more than 10 million triangles can be fitted into the GPU memory. Finally, Aila and Laine showed that GPU ray tracing does not have to be memory bound. They achieve impressive results with their ray tracer.

3.2 Final thoughts

These days are very exciting times in computer graphics. Ray tracing does not seem to be able to replace the rasterization pipeline. Therefore the speed disadvantage issue is much too large. But with upcoming new real time global illumination ideas[24][34], ray tracing receives more and more attention. The first results look overwhelming and add a huge leap forward in realism. Which one of the presented approaches will be used in high performance real time applications remains to be seen. Also currently used techniques like ambient occlusion and (soft-) shadows will benefit from ray tracing. If current GPUs continue with their transistor doubling every year they will soon have the power to do tasks like ray tracing in acceptable times.
Bibliography


[18] Intel. URL http://www.intel.com/


[22] OpenCL Khronos. URL http://www.khronos.org/opencl/


[27] NVIDIA. Interactive ray tracing with cuda. URL http://developer.nvidia.com/object/nvision08-IRT.html


29


